

## Lec. slide 1. Micro

What is Embedded system?

↳ Combination of hardware and software.

↳ Designed to perform specific function.

↳ combination of computer hardware with the software to control, monitor, communicate to do multiple operations

ex: washing machine, Digital camera, DVD Player.

General Purpose system vs. Embedded system

General Purpose	Embedded.
↳ combination of <del>any</del> <sup>generic</sup> HW & general purpose OS for executing variety of apps.	↳ combination of special purpose <del>OS</del> HW and Embedded OS for executing specific set of apps.
Need not to be deterministic in execution behavior	Execution behavior is deter. for certain types of embedded system.
Example: Computer, laptop	Example: DVD, Printers.

## Characteristics of Embedded systems:

- \* Single Functioned.
- \* Reliability.
- \* Cost effectiveness
- \* low power Consumption
- \* Fast execution time
- \* efficient use of memory.
- \* Processing Power.

## \* Major application areas of Embedded system:-

- 1) Consumer electronics (Cameras, etc)
- 2) Household appliances (TV, DVD Players, Washing machine)
- 3) Home automation & security system (Fire alarms -- etc)
- 4) Telecom (handset, Cellular telephones -- etc)
- 5) Healthcare (scanners, ECG, etc)
- 6) Banking & retail (ATM, currency counter, etc)

## Categorization of embedded system

### [1] Based on Generalization:-

a. First generation (from <sup>4</sup>8 to 8 bit microcontrollers & ex: stepper motor control units)

b. 2nd generation (8 to 16 bit micro controllers & ex: sensors)

c. 3rd " (16 to 32 bit & ex: DSP)

d. 4th " (Smart phones, Mobile internet devices, etc)

[2]

[2] Based in complexity & Performance :-

a. Small scale Embedded system

↳ small Processing elements with simple Program

↳ low cost.                      ↳ lower Performance.

b. Medium scale ES :-

↳ slightly complex in HW & SW.

↳ low cost.                      ↳ Medium Performance.

c. Large scale ES

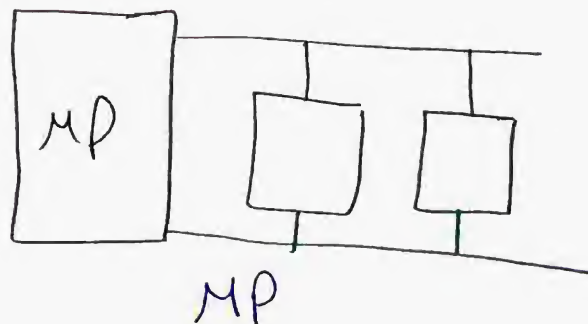
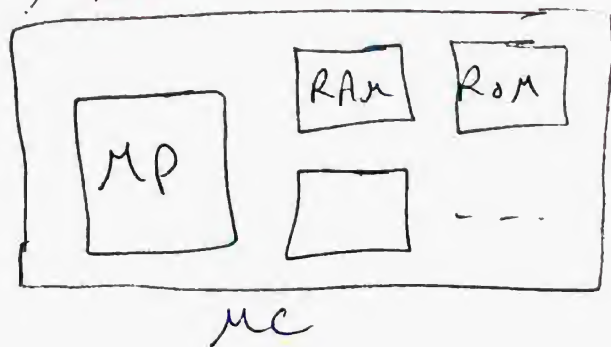
↳ complex                      ↳ High Performance.                      ↳ low cost

[3] Based on deterministic behavior.

↳ Hard real time                      ↳ soft real time.

[A] Based on Triggers (Event based & time based)

Micro Controller vs Micro Processor



~~MC~~ ES → uses (MC) to do specific task.

PC → uses MP to do different things.

[3]



## \* Criteria of choosing MC

- \* Speed
- \* Power Consumption
- \* Cost
- \* RAM & ROM
- \* I/O Ports
- \* Packaging (Put MC in which Package)
- \* Community

## Components of 8051 MC

- a) 128 bytes RAM
- b) 4K ROM
- c) 2 timers
- d) one serial Port
- e) 6 interrupt
- f) 4 I/O Ports (each 8 bits)

## Notes

- 1) 8051 is subset of 8052
- 2) 8031 " a ROM-less 8051

	8051	8052	8031
ROM	4K	8K	0K
RAM	128	256	128
Timers	2	3	2
I/O Pins	32	32	32
Serial Port	1	1	1
Interrupt Sources	6	8	6

- 3) \* vast majority of 8051 register are 8-bit registers.



## Micro Processor V-s Micro Controller

Micro Processor	Micro Controller
→ use CISC Architecture	→ use RISC and Harvard.
→ Has ROM, RAM, secondary storage memory, I/O etc Placed on board and Connected through Buses.	→ ROM, RAM, I/O PrePherals are Combined in single integrated circuit
→ Less secured	→ more secured.
→ expensive	→ cheaper
→ Design takes more time	→ Design it takes <del>less</del> less time.

RISC	CISC
Simple instruction Format	Variable inst. Format
Large set of CPU registers	Small set of General Purpose registers
very few addressing modes	Large number of addressing modes.
Simple Pipelining	Complex Pipelining
supports on chip cache memory	seldom supports on chip cache memory

## 8051 Assembly language Programming

1) How instructions are executed.

2) Registers.

- a) 8 bit registers Me
- b) A: Accumulator, must be destination of any operation.
- c) B, PC, DPTR, SP
- $R_0 - 7$

Any Assembly Program { instructions & directives }

instructions  $\Rightarrow$  [label:] opcode [operands] [; Comments]

Directives (not instructions)

↳ but they help assembler in assembly Process.

[ex] ORG address  $\Rightarrow$  PC = address

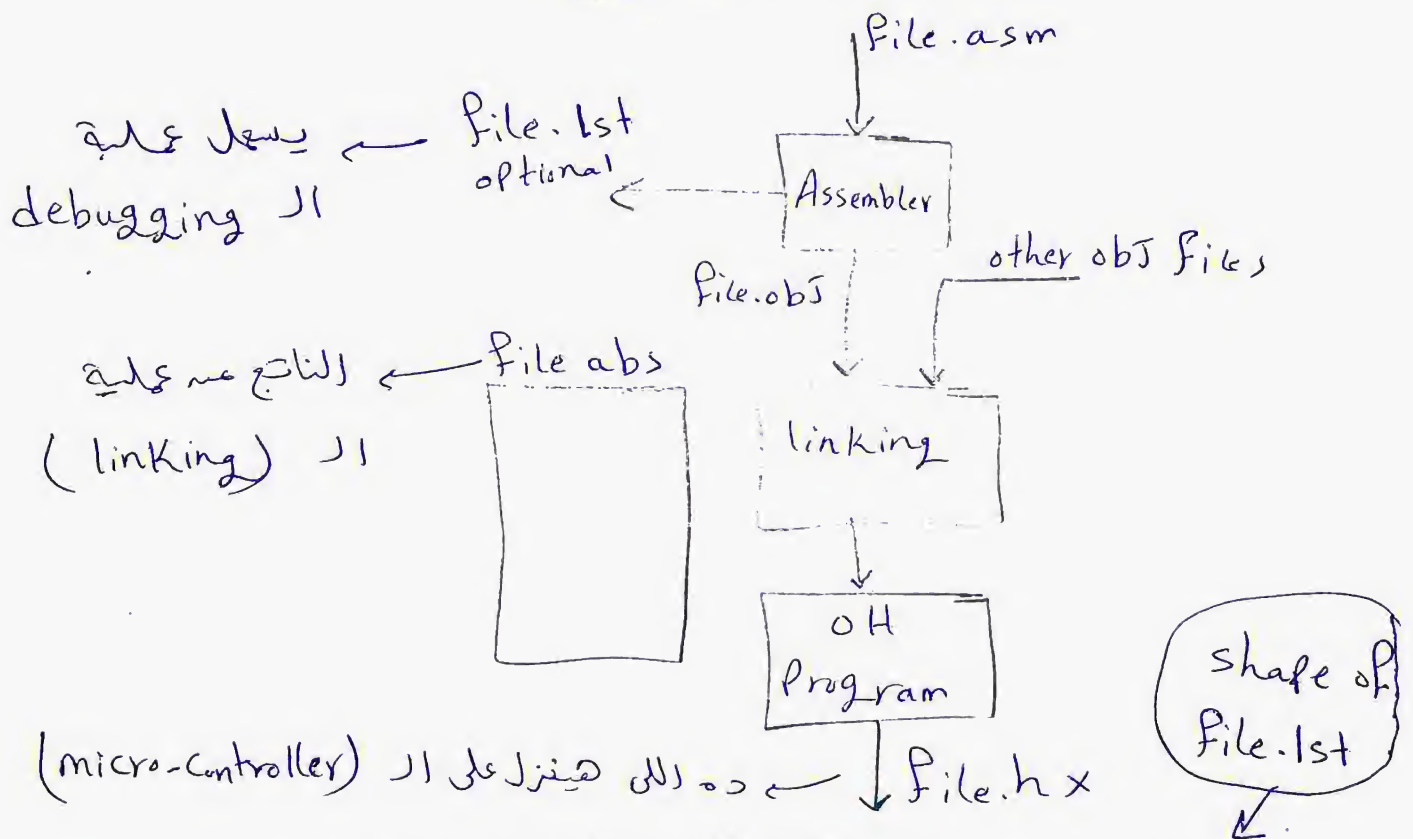
له معناها انه البرنامج بتاعي اول (instruction) هيقع موجود  
في ال (Address) اللي عنده ده وهو اللي ال (Program Counter)  
هيشارة عليه في الاول.

→ END

يعرف ال (Assembler) انه دي نهاية الجزء اللي بيعمل  
ليه (Assembly).

## step of assembly language Program

(File.asm) يكتب فيه ار (instruction) الى عايزها  
 له بيخد على ار (assembler) يطلع (file.object)  
 وده التالى الى انتا محتاج يحصل له (linking) مع  
 الحاجة الى انتا عايز تعلقها .



Address	Machine	Assembly	Comments
0000	7D25	ORG 0000	
0002	7F34	MOV R5, #25	
0004	2D	MOV R6, #34	
0005	—	ADD A, R5	



من أول ما تفتح الـ (Program) هتلاقى لـ

الـ PC عندك فيه  $PC = 0000$  وكل

شوية عمل (increment) وهيقعاري

لـ الـ الـ جاي مش (instruction)

0000	7D
0001	25
0002	7F
0003	34
:	:

RAM

ملحوظة: المقدرة أكتب برنامج السعة بتاعته أكثر من

(4K byte) لأن سعة الـ RAM ← 4K byte

(2) هيدلـ برنامج (assembly) ديطلب سلة تصلا الجدول

بتاع الـ (lst. file) في الـ الصفحة السابقة وبعدين تعمل الجدول في

الـ الـ الصفحة دي رى مثال في مـ 75/74 في (slides)

### DB directive

↳ most widely used ↳ used to define 8-bit data

شركة لو عاين أطبع (Hello world) بار (assembly)

لـ محتاج تكتب موجود في الـ (memory)

ORG 500

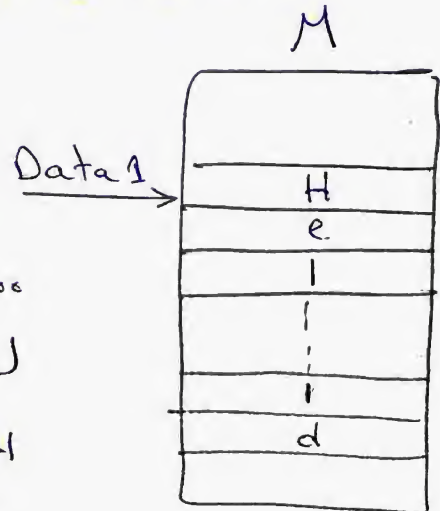
Data 1 DB "Hello world"

لـ عند الـ (label Data 1) الـ الـ حبيزة عـ 500  
كل ما بقول الـ بيشار على الأمر ..

لـ أنا كده عرفت بار (DB) مجموعة الأوامر

الموجودة من أول الـ (address 500)

محمولة لـ (Hello world)



## \* EQU

له بديل (define) لثابت من غير ما يلاحظ مكانه في ال (memory)  
 له بديل في ال (data label) وادلما ال (label)  
 يظهر في البرنامج يستخدم قيمة الثابت خوراً.

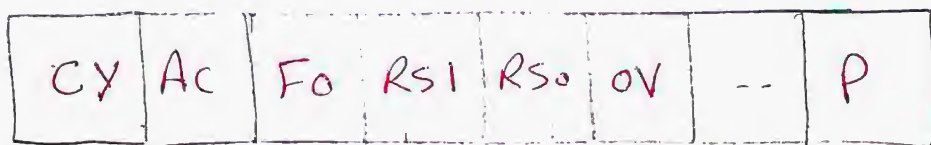
ex

~~Count~~ EQU 25 → use EQU For Counter Constant

Mov R3, #Count → Constant used to load R3.

Flag register (PSW) → Program status world

له عبارة عن (8-bit register) يستخدم 6-bit



↳ not used

Where

CY → Carry. Ac → Auxiliary carry.

RS0, RS1 → Bank selectors. OV → overflow.

P → Parity Flag. Fo → user definable.

ex

$$\begin{array}{r}
 00111000 \\
 00101111 \\
 \hline
 01100111 \\
 \hline
 6 \quad 7
 \end{array}$$

$$\begin{array}{r}
 38 \\
 + 2F \\
 \hline
 67
 \end{array}$$

P=1, Ac=1, CY=0

عدد البتات  
 P=1 ← odd  
 P=0 ← even  
 CY=0 ← 0=D7  
 انتقلت من D3 ← D4 ومعاك  
 Ac=1 ← carry

\* RAM : 128 bytes

له مكان اخذ و فني ال (Variables)

اللي بيجعل فني (operations)

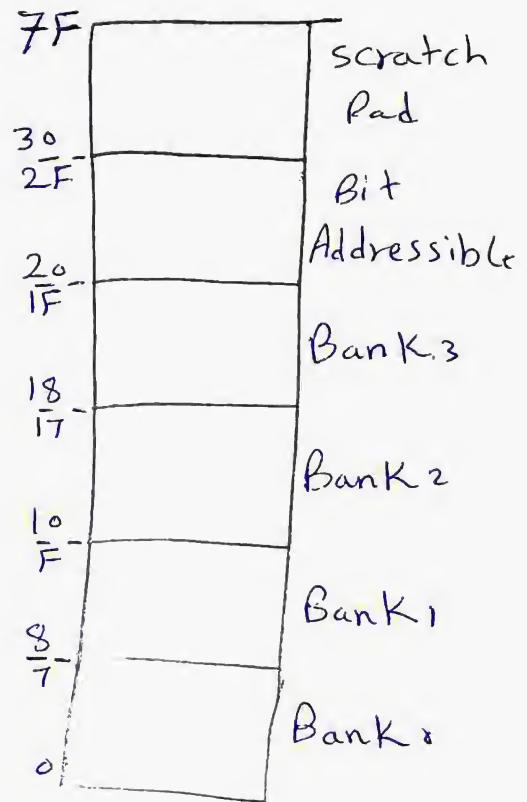
↳ (RS1, RS0)

حسب قيمه بختار ال (Bank)

اللي ال (register) بتاعي بيتعامل

معاه .

RS0	RS1	Bank
0	0	0
0	1	1
1	0	2
1	1	3



(stack) (default) بتاعه ده ال (Bank 1)

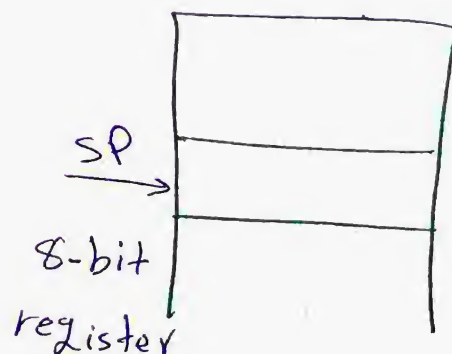
↳ push & pop

Default

SP = 7

عنه ال (Bank 1) بتاعي

(8) نه





Push → storing CPU register in stack  
 → increment then push.

Pop → loading contents of stack back into  
 CPU register.  
 → pop then decrement.

ملحوظة: لا يشترط أن  $(SP=7)$  هو فقط بيشار على  
 عنوان معين ويمكن العنوان الذي بعده هو الذي يتخزن فيه  
 الـ (data) باعتبار.

**[Ex]** show stack and stack pointer

Mov R6, #25H

Mov R1, #12H

Mov R4, #0F3H

Push 6

Push 1

Push 4

**[Sol]**

after Push 6

0B	
0A	
09	
08	

SP=7

0B	
0A	
09	
08	25

SP=08

Push 1

0B	
0A	
09	12
08	25

SP=09

Push 4

0B	
0A	0F3H
09	12
08	25

SP=0A

**[u]**

## JUMP, Loop and CALL instructions

Page 104

Sec 6

Loop repeat sequence of instructions a certain number of times.

is performed by  $\Rightarrow$  DJNZ Reg, label

بعد (decrement) لا (reg) و (check) كل شوية  
، طالما ليس بهنجر يعمل (Jump) لا (label)

ex Reg = value

بعد (decrement) لا (reg) وطول

ما هو ليس بهنجر يعمل (loop)

DJNZ Reg, loop

ماذا لو هحتاج أكرر (action) أكثر من 256 مرة

له أكبر قيمة لا (reg) هي 255 (FF) لكنه لو عندي (loop)

تحتاج قيمة أكبر من (255) فعل أكثر من (loop) جمعه  
يعني (nested loop)

لم يعني مثلاً لو عندي 700 فعل (2 loops) واحدة 10  
والأخرى 70 ، والناتج يكون حاصل ضربهم

## Jump

### Conditional Jump:

لـ بعد (operation) تم العمل (check flag) وعلى أساسه يعرف أخذ أي (action) في البرنامج.

ex JZ label

لـ يعمل (check) على (zero flag) ومنها العمل (Jump) لـ (label)

if equal to zero  $\Rightarrow$  jump to label.

else

(unconditional) في المنطقة \* عمل (label 2) و (Jump)

ex  
JZ label 1  
\*  
label 1  
label 2

## Jump

### 1) LJMP (long Jump)

لـ بعد (Jump) لأي مكان في البرنامج.

لـ ثلاثة بايتات (3 bytes) واحدة لـ (opcode) ،  
اثنين لـ (address)

### 2) SJMP (Short Jump) (-128 to 127)

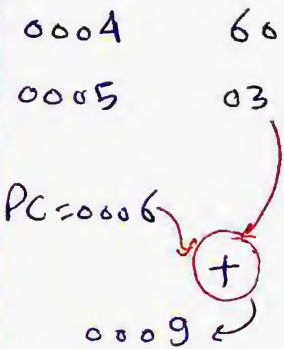
لـ بقدر العمل (Jump) فوقه ارتحت بعد أقصر (128 bit)  
لـ بايتات (2-byte)



# Page 112 ex

Line	Pc	oPCode	operand
01	0000		ORG 0000
02	0000	7800	MOV R0, #0
03	0002	7455	MOV R1, #55H
04	0004	6003	JZ NEXT
05	0006	08	INC R0
06	0007	04	AGAIN: INCA
07	0008	04	INCA
08	0009	2477	NEXT: Add A, #77H
09	000B	5005	JNC OVER
10	000D	E4	CLR A
11	000E	F8	MOV R0, A
12	000F	F9	MOV R1, A
13	0010	FA	MOV R2, A
14	0011	FB	MOV R3, A
15	0012	2B	OVER: ADD A, R3
16	0013	50F2	JNC AGAIN
17	0015	80FE	HERE: SJMP HERE
18	0017		END

ex



صاحب جیب (oPCode)  
لا loop او jump

عند JNC OVER  
عائز يطلع ب 5005  
قبيل طرح (0012) مكان  
ما ار (OVER) راحت  
من ار (0D) ار PC

كذلك مع  
JNC AGAIN

لوار (oPCode)  
ممكن من رقمين ار  
PC هيزيد ب (2)  
ولو رقم واحد ار  
(PC) هيزيد (1)

## Call

هو يكتب شوية أكواد وفي النهاية عايز أعمل  
(subroutine) فيعمل فيه (Call)

\* لا يتعمل (Jump) بتكمل الكود بتاعه  
لكنه مش بتراجع للبداية .

لكنه في ال (Call) بتستخدم أمر  
(return) عشان ترجع للبداية .

ex

====

Call label1

====

Sub1:

====

RET return

هو ال (Call) بيعمل Push لا (PC) ويعمل تغيير فيه  
ر (Sub1) عشان يشاور عليه

Push PC

change PC to sub1

Sub1:

====

RET

=> POP PC

قبل الجزء بتاع (Sub1)

هضع (HLT) معناه

نهاية البرنامج

## Call

لما بعمل (Call) في أي مساحة في البرنامج (Long call (Lcall)

لما ال (subroutine) موجود في أي مكان  
خلاف ال (64Kbyte) .

لما يعتبر (3-byte instruction)

2) Acall (absolute call)

(2K-byte) فقط

لما بعمل (Call) في صدد

لما (2-byte instruction)



## Notes

له لو في ار (main) دار (Subroutine) بتستعمل نفس

ار (registers) ← ار (Call) مش معن بانه يعنى ار

ار " لأنه بيتعامل مع ار (Push, return) فقط.

ل في هذه الحالة قبل بداية ار (Subroutine) عمل Push

لقيم ار (registers) اللى هستخدمها وفي الآخر عمل (Pop) ليهم  
بس بالعكس.

له لو علت (Push) ودمفيسر في الآخر (Pop)

مش صيالح قيم ار (PC) المطلوبة.

له لازم كل (Push) في البداية يقابلها

نفس ار (Pop)

Push 4

Push 5

Pop 5

Pop 4

\* ار (Instruction) يتم تفرقة بال (Crystal oscillator)

ار machine cycle : هي (cycle) يتم تنفيذ الامر فيها

له ار (microcontroller) يحتوى على مجموعة من ار (operations)

كل (operation) تحتاج كام (cycle)

ار (machine cycle) ل (atmel) عبارة عن (12 clocks)

→ Find Period of machine cycle for 11.0592 MHz  
crystal frequency.

$$\frac{11.0592}{12} = 921.6 \text{ KHz} \quad \& \quad \text{machine cycle} = \frac{1}{921.6 \text{ KHz}} = 1.085 \mu\text{s}$$

16



## Addressing modes

1) Immediate

2) register.

3) Direct

4) register indirect

5) Indexed.

↑  
↑  
↑  
Accessing memories.

### 1) Immediate

↳ src. operand is constant. `Mov A, #25H`

↳ load info. into any registers, including 16-bit DPTR register.

### 2) Register

↳ hold data to be manipulated (`Mov A, R0`)

↳ src. & dst. registers must match in size

ex `Mov DPTR, A (xx)` & `Mov DPTR, #25F5H`

↳ `Mov R4, R7 (xx)`

### 3) Direct addressing mode

↳ used to access RAM locations 30-7FH

ex: `Mov A, 4` & `Mov A, R4`

no # sign here

\* SFR (special function register)

↳ can be accessed by their names or their addresses

↳ have addresses

between 80H

and FFH.

`Mov 0E0H, #55H`

`Mov A, #55H`

`Mov 0F0H, R0`

`Mov B, R0`

**[Ex]** write code to send 55H to ports P1 & P2 using.

a) their names

Mov A, #55H

Mov P1, A

Mov P2, A

b) their addresses

Mov A, #55H

Mov 80H, A

Mov 0A0H, A

---

\* Stack and direct addressing mode

↳ only direct addressing mode is allowed for pushing or popping the stack.

↳ Push A (invalid)  $\Rightarrow$  it should be Push 0E0H

**[Ex]** show the code to push R5 and A onto the stack and then pop them back into R2 & B, (B=A, R2=R5)

**[sol]**

Push 05 ; Push R5 to stack

Push 0E0H ; Push register A to " .

Pop 0F0H ; Pop top of stack to B  
now B = A

Pop 02 ; Pop top of stack into R2  
now R2 = R5

#### [4] register indirect addressing mode:-

↳ register is used as a pointer to data (only R0, R1)

MOV A, @R0 → move contents of RAM whose address is held by R0 into A

MOV @R1, B

[EX] write a program to copy value 55H into RAM memory locations 40H to 41H using:

a) direct addressing mode

MOV A, #55H

MOV 40H, A

MOV 41H, A

b) register indirect without loop

MOV A, #55H

MOV R0, #40H

MOV @R0, A

INC R0

MOV @R0, A

c) register indirect with loop

MOV A, #55H

MOV R0, #40H

MOV R2, #02

AGAIN: MOV @R0, A

INC R0

DJNZ R2, AGAIN

↳ advantage of register indirect that it makes accessing data dynamic rather than static as in direct addressing mode.



**EX** write Program to clear 16 RAM locations starting at RAM address 60H

s.l

~~clear~~ clear A ; A = 0  
Mov R1, #60H ; R1 = 60H pointer  
Mov R7, #16 ; Load counter R7 = 16  
AGAIN : Mov @R1, A ; clear RAM R1 points to  
INC R1 ;  
DJNZ R7, AGAIN ; loop until counter = zero

---

EX: write Program to copy a block of 10 bytes of data from 35H to 60H

solution

Mov R0, #35H  
Mov R1, #60H  
Mov R2, 10  
Back : Mov A, @R0  
Mov @R1, A  
INC R0  
INC R1  
DJNZ R2, Back

## Indexed addressing mode

↳ widely used in accessing data elements of look-up table entries located in Program ROM.

instruction used  $\Rightarrow$  `MOVC, @A+DPTR`

↳ code      ↳ 16-bit register

~~content of P~~

Note: look for ex (5-6 & 5-8) in slides Page 175, 176

### Notes

→ In many apps, size of Program Code doesn't leave any room to share 64K-byte code space with data.  
↳ 8051 has another 64K-bytes of space set aside for data storage.

↳ 8051 has 128K-bytes of memory space  
↳ 64K bytes of code + 64 bytes of data

→ data space between code & data cannot be shared.

→ In many apps. we use RAM locations 30-7FH as scratch pad.

\* write Program to toggle

P1  $\rightarrow$  200 times. use

RAM location 32 to hold your counter value instead of

R0-R7 register

`MOV P1, #55H`

`MOV 32H, #200`

`LOP1: CPL P1`

`ACALL DELAY`

`DJNZ 32H, LOP1`

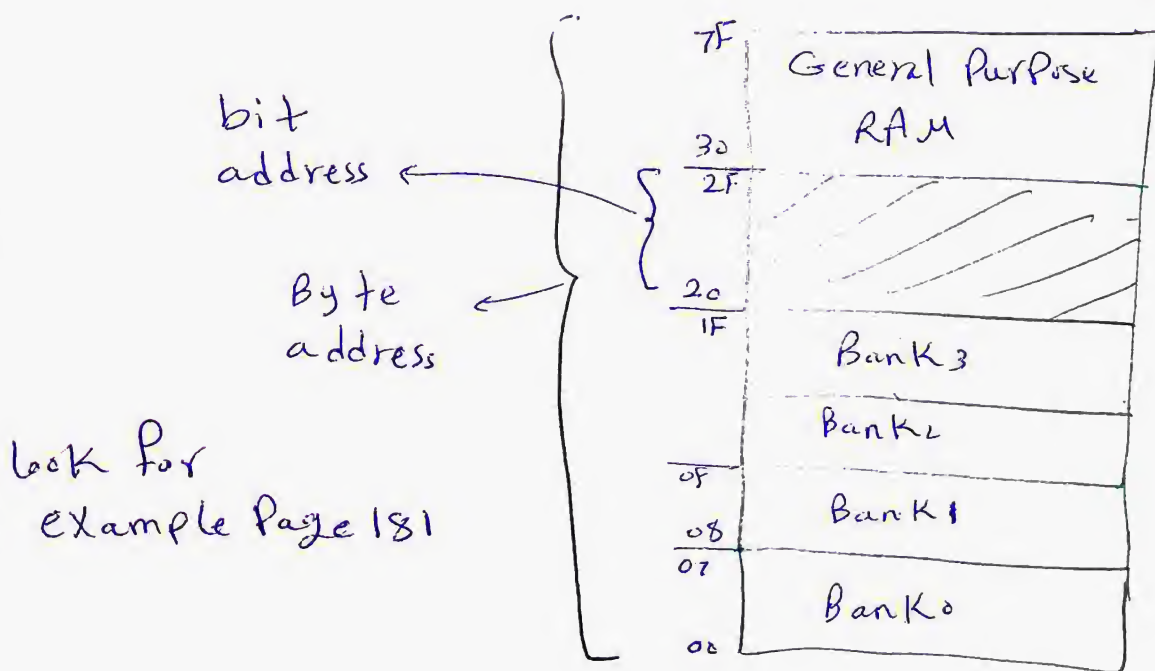
## Bit Addresses

⇒ Many microprocessors allow program to access registers and I/O ports in byte size.

↳ 8051 is single-bit operation ⇒ allow programmer to do operations set, clear, move, complement bits of port or memory or register.

ROM → only not bit-addressable.

\* Internal RAM locations 20-7FH are (byte & bit) addressable  
↓  
00-7FH



↳ 00-7FH can be accessed in byte size using direct & ~~indirect~~ register indirect addressing mode.

→ Instructions here are single bit likies:

JB bit, Target if bit = 1 → Jump to target

JBC bit, target " " " → " " " then clear bit.



Ports  $P_0 - P_3 \Rightarrow$  bit addressable

\* when we access Port in single bit

SETB X.Y      X  $\rightarrow$  Port number ( $P_0, P_1, P_2, P_3$ )  
Y  $\rightarrow$  desired bit no. from 0 to 7

registers  $\Rightarrow$  A, B, PSW, IP, IE, ACC, SCON & TCON  
↳ are bit addressable.

EX write Program to save accumulator in R7 of bank 2.

CLR PSW.3

SETB PSW.4

MOV R7, A

look for examples  
at page 188, 189

\* BIT Directive used to assign bit-addressable  
I/O and RAM locations.

EX switch is connected to Pin  $P1.7$  and LED to Pin  $P2.0$  write a program to get status of switch and send it to the LED

LED      BIT      P1.7

SW      BIT      P2.0

HERE:    MOV    C, SW

          MOV    LED, C

          SJMP   HERE

EQU in bit addresses : assign addresses

- ↳ Defined by names (like P1.7)
- ↳ " " addresses (~~97H~~ like 97H)

→ Example at Page 192.

\*8052 has another 128 bytes of on-chip RAM with addresses 80-FFH → called upper memory.

MOV @R0, A & MOV @R1, A

→ example Page 194

### Arithmetic & Logic instructions and Programs

\* Addition of unsigned numbers:- Add A, src

↳ Destination is always in register A.

↳ src. operand can be register, ~~inter~~ immediate data or in memory.

↳ memory-to-memory arithmetic operations not allowed in 8051.

\* Show How Flag register is affected by this:

MOV A, #0F5H ; A=F5

ADD A, #0BH ; A=F5+0B = 00

	<u>Sol</u>	
F5H	1111 0101	
0BH	0000 1011	
<u>100H</u>	0000 0000	

{ CY = 1  
PF = 1 (no. of 1s zero)  
AC = 1 → carry from D3 to D4

## Addition of individual bytes

→ Assume that RAM locations 40-44H have these values.  
write program to find sum of values. at the end of  
program, register A should contain low byte & R7 the high byte.  
40 = (7D) , 41 = (EB) , 42 = (C5) , 43 = (5B) , 44 = (30)

Sol

MOV R0, #40H ; load Pointer

MOV R2, #5 ; counter

CLR A ; A = 0

MOV R7, A ; clear R7

AGAIN : ADD A, @R0 ; ~~add byte ptr to~~

JNC NEXT ; if CY = 0, don't add carry

INC R7 ; keep track of carry

NEXT : INC R0 ; increment pointer

DJNZ R2, AGAIN ; repeat until R2 = 0

## Note

↳ when adding two 16-bit data operands, Propagation of carry from lower byte to higher byte is concerned.

Example at 198.

BCD Binary Coded decimal

↳ binary representation of digits 0 to 9

BCD { ~~packed~~ Packed BCD  
unpacked.



## 1] unpacked BCD

↳ lower 4 bits of number represent BCD number + rest of bits are zero

[EX] 00001001 → unpacked BCD for 9.

## 2] Packed BCD

↳ single byte has two number (one in lower 4-bits and one in the upper 4 bits)

[EX] 01011001 → is packed BCD for 59H

↳ Adding two BCD numbers must give BCD result.

[EX]  $\Rightarrow$  result = 3FH  $\rightarrow$  it <sup>should</sup> be 17+28=45  
↳ correction 3F+06=45H  
MOV A, #17H  
ADD A, #28H

DA: decimal adjust for addition

$\Rightarrow$  After ADD or ADDC instruction

1) if lower (4 bits)  $\geq 9$  or AC = 1 add (0110) to lower 4-bits.

2) if upper nibble (4-bits)  $\geq 9$  or CY = 1 add (0110) to upper 4 bits.

[EX]

29	0010 1001	
18	0001 1000	
41	0100 0001	AC = 1
6	0110	
47	0100 0111	

Example at Page 203 (look for A)

Add Program to find sum of numbers. result must be in BCD.

40 = (71), 41 = (11), 42 = (65), 43 = (59), 44 = (37)

<p><del>MOV</del> <span style="border: 1px solid black; padding: 2px;">S.I</span> Mov R0, #40H</p> <p>Mov R2, #5</p> <p>CLR A</p> <p>Mov R7, A</p> <p>AGAIN: ADD A, @R0</p>	<p>DA A <math>\rightarrow</math> adjust BCD</p> <p>JNC NEXT</p> <p>INC R7</p> <p>NEXT: INC R0</p> <p>DJNZ R2, AGAIN</p>
---	---

In micro Processor: Subtraction

SUB

SUBB  $\rightarrow$  sub with borrow

in 8051  $\Rightarrow$  only SUBB

SUBB A, src  $\Rightarrow A = A - \text{src} - \text{CY}$

$\rightarrow$  to get SUB from SUBB  $\Rightarrow$  Put CY = 0

~~In this case~~ SUBB when CY = 0

1) take 2's complement of src. operand.

2) Add it to A

3) invert the carry.

AC	0100 1100	0100 1100	
- 6E	0110 1110	$\xrightarrow{2's}$ 1001 0010	+
<hr/> -22	<hr/>	<hr/>	
	CY=1	01101 1110	
	<span style="border: 1px solid black; padding: 2px;">27</span>		

\* SUBB when CY=1

used for multi-byte numbers & will take care of borrow of lower operand.

$$2762H - 1296H = 14CCH$$

8051 supports byte by byte multiplication

MUL AB ;  $A * B$

ex: Page 207

Division

DIV AB ;

$A/B$

$$\frac{95}{10}$$

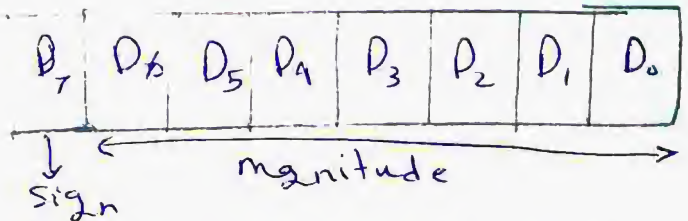
$\Rightarrow$  09  $\rightarrow$  quotient  
05  $\rightarrow$  remainder

look for example at Page 209

Signed operands

two numbers  $\Rightarrow$  0 to +127

-ve " (2's complement)



overflow (happens if result of operation on signed numbers is too large for register.)

ex

$$\begin{array}{r} 96 \\ + 70 \\ \hline 166 \end{array}$$

$$\begin{array}{r} 0110 \ 0000 \\ 0100 \ 0110 \\ \hline 1010 \ 0110 \Rightarrow OV = 1 \end{array}$$

28



\* In 8-bit signed numbers ~~add~~<sup>operations</sup>, OV set to 1 if either occur:

1) there is carry from  $D_6$  to  $D_7$ , but no carry out of  $D_7$  ( $CY=0$ )

2) there is carry from  $D_7$  out ( $CY=1$ ), but no carry from  $D_6$  to  $D_7$ .

### notes

1) in signed number addition, we must monitor status of  $CY$ . use JNC or JC instructions.

2) in signed no. add, OV Flag must be monitored.

JB PSW.2 or JNB PSW.2

$\Rightarrow$  2's complement of no.  $\Rightarrow$  CPL A  
ADD A, #2

AND instruction (ANL)	X	X	S
$\rightarrow$ destination is Accumulator	0	0	0
$\rightarrow$ src can be register, memory or immediate	0	1	0
	1	0	0
	1	1	1

EX

MOV A, #35H

ANL A, #0FH

## OR operation (ORL)

**[EX]** Mov A, #04H

ORL A, #68

$$\begin{array}{r}
 0000 \ 0100 \\
 0110 \ 1000 \\
 \hline
 0110 \ 1100
 \end{array}$$

X	<del>Y</del>	S
0	0	0
0	1	1
1	0	1
1	1	1

## XOR operation (XRL)

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

## Compare instruction (CJNE dst, src, rel.addr.)

↳ Compare two operands, Jump if they are not equal.

↳ dst. is Accumulator or one of Rn registers.

↳ changes CY to see if dst. is larger or smaller.

**[EX]** CJNE R5, #80, NOT\_EQUAL

NOT\_EQUAL:

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

note in CJNE instruction, any Rn registers can be compared with immediate value.

↳ Comparing is subtraction except that operands remain unchanged.

EX: at Page 224

Rotate instruction RR - A rotate right



[EX] 0011 0110  $\Rightarrow$  0001 1011

\* RL A rotate left

[EX] 0011 0110  $\Rightarrow$  0110 1100

\* Rotating through carry ~~RR~~ ~~RRCA~~

[1] RRC A

CX=0  $\Rightarrow$  ~~0010 0110~~ 0010 0110  $\rightarrow$  0001 0011

CX=1  $\Rightarrow$  0001 0011  $\rightarrow$  0000 1001

[2] RLCA (same but left)

~~RRCA~~



## Serializing data

↳ sending byte of data one bit at a time through a single pin of microcontroller.

using serial port: to transfer one bit at a time & control sequence of data and spaces between them.

examples at Page 230, 231 | table of carry operations at Page 232, 233

## Swap

↳ It swaps the lower and higher nibbles

before: D7-D4 D3-D0  $\Rightarrow$  D3-D0 D7-D4

FX    `MOV A, #72H`    ; A = 72H  
      `SWAP A`        ; A = 27H

## notes

\* DS5000T microcontrollers have real-time clock (RTC)

\* To convert ASCII to Packed BCD

~~ASCII~~  
Key    ASCII    unpacked BCD  
4       34       0000 0100  $\Rightarrow$  0100 0111 or 47H  
       $\Rightarrow$          $\Rightarrow$   
7       37       0000 0111

~ example at Page 239 & 240

note  
To ensure the integrity of ROM contents, every system must perform checksum calculation

checksum  
↳ detects any corruption of contents of ROM.

↳ used checksum byte

To calculate checksum byte of series of bytes

↳ Add bytes together & drop carries.

↳ take 2's Complement to total sum, it becomes last byte of series.

To perform checksum operation, add all bytes including checksum byte.

↳ result must be zero.

↳ if not zero, one or more bytes of data have been changed.

**EX)** For these 25H, 62H, 3FH, 52H

a) Find checksum byte

$$\begin{array}{r} 25H \\ 62H \\ 3FH \\ 52H \\ \hline 118 \end{array}$$

drop carry you get 18

2's complement " " E8H

b) Perform checksum to ensure integrity

$$\begin{array}{r} 25H \\ 62H \\ 3FH \\ 52H \\ E8H \\ \hline 200 \end{array}$$

drop carry 00

# 8051 Programming in C

## Why Program in C

- 1) less time consuming but larger hex file size
- 2) easier than assembly.
- 3) easy to modify & update.
- ④ Can use code available in function libraries.

## Data types

1) unsigned char :: 8-bit at range 0-255

↳ ~~widely used~~

↪ C Compilers use signed char as default if we don't put the keyword unsigned

Examples at Page 249 & 250

[2] signed char (8-bit data type)

↳ use MSB D7 to represent - or +

↳ Give values from -128 to +127.

\* Example at Page 251

\* single Bit (sbit)

↳ allow access to the single bits of SFR registers.

Example at Page 253



## \* Bit & sfr

↳ bit datatype allow access to single bits of bit-addressable memory spaces 20-2FH.

↳ to access byte-size SFR registers, use sfr datatype.

Time delay to create it:

1) using 8051 timer.

2) using simple for loop

**[Ex]** write 8051 program to toggle bits of P1 continuously forever with some delay.

**[sol]**

```
#include <reg51.h>
void main (void)
{
    unsigned int x;
    for ( ; )
    {
```

```
        P1 = 0x55;
        for (x=0; x<40000; x++),
            P1 = 0xAA;
        for (x=0; x<40000; x++);
    }
}
```

toggle P1 bits of P1 Port with delay 250 ms

```
#include <reg51.h>
void MsDelay (unsigned char)
void main (void)
{
    while (1)
    {
        P1 = 0x55;
        MsDelay (250);
```

```
        P1 = 0xAA;
        MsDelay (250);
    }
}

void MsDelay (unsigned int i, j)
{
    unsigned int i, j;
    for (i=0; i<i*time; i++);
    for (j=0; j<1275; j++);
}
```

See examples From 258 to 267

---

operators

AND(&&) OR(11) NOT(!)

shift right(>>) , inverter(~) EX-OR(^)

See examples From 269 to 278

---

\* 8501 C compiler allocates RAM locations

1) Bank 0 (0-7)

\* 2) Individual variables.

3) Array elements

4) stack

examples 279, 280

---

\* extra 128 bytes of RAM helps 8051/52<sup>C</sup> compiler to manage its registers & resources <sup>more</sup> effectively.

\* To make C-compiler use code space instead of RAM space, put Keyword "code" in front of variable declaration as in ex: Page 283.

at Page ~~284~~ 284 & 285 → discussing Programs  
(may be important)



# Interrupts chapter

Interrupt: external or internal event that interrupt microcontroller to inform it that a device needs its service.

\*Single microcontroller can serve devices by:

## \*Interrupts

↳ whenever any device need its service, it notifies the microcontroller by send it interrupt signal

↳ then MC stop whatever it does, and serve the device.

## \*Polling

↳ MC continuously monitors the status of a given device

↳ when conditions met, it performs the device.

↳ then, it moves on to monitor next device until serve all devices.

## Advantage:

↳ MC can serve many devices

## Disadvantage

↳ Can't serve all devices at the same time

## Advantage

↳ can serve all devices if conditions are met

## Disadvantage

↳ not efficient, cause it wastes time to MC, that not need service.

## \*Interrupt service routine (ISR)

↳ For every interrupt, there is fixed location in memory that hold address of its ISR

↳ program associated with interrupt.



\* Steps in Executing interrupt :- MC go in these steps:

- 1) Finishes instruction it is executing and saves address of next instruction on stack.
- 2) Saves current status of all interrupt internally (not in stack)
- 3) It Jump to a Fixed location in memory, called interrupt vector table, that holds address of ISR.
- 4) MC gets address of ISR and jump to it.  
↳ starts to execute ISR until reaching last inst. of Subroutine (RETI : return from interrupt)
- 5) upon executing RETI inst., MC returns to the place where it was interrupted

Six interrupts at 8051

- 1) reset & power-up reset
- 2) Two for timers
- 3) Two for Hw external interrupts (P3.2 & P3.3)
- 4) serial communication has single interrupt that belong to ~~the~~ receive & transfer.

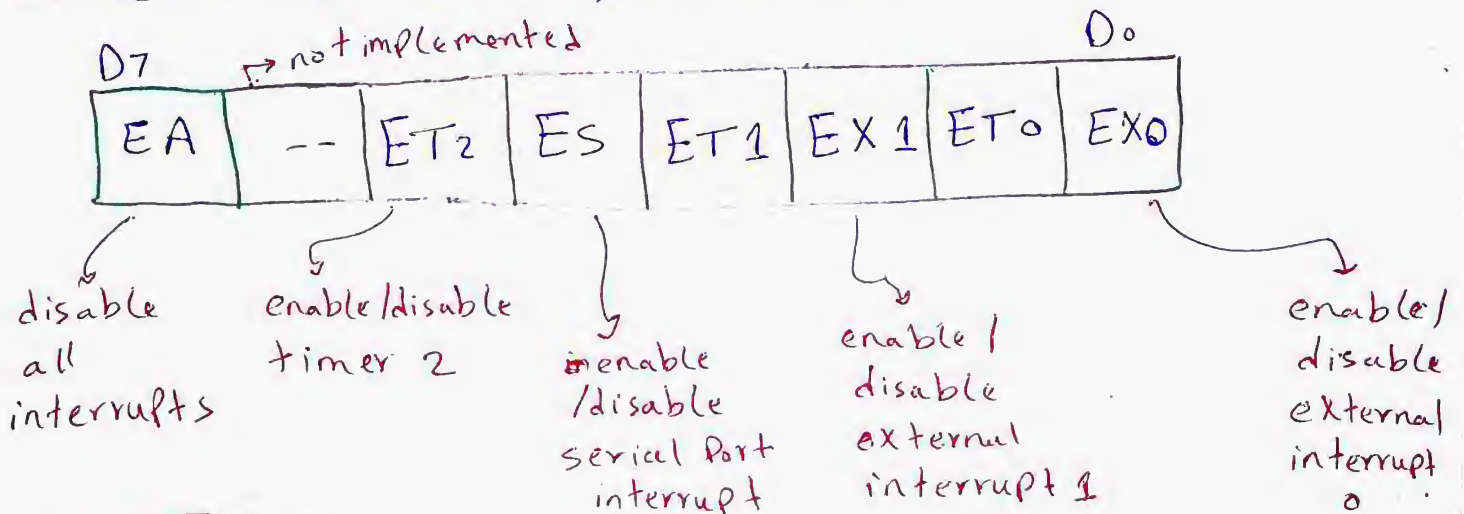
INT0, INT1  
↑

## P. 427 Enable & Disable interrupt

\* upon reset, all interrupts are disabled.

\* interrupts must be enabled by SW so that microcontroller could respond to them.

IE: (Interrupt enable) → enable & disable interrupt



### Steps of enabling an interrupt

1) Bit D7 must be set to high

2) value of EA

↳ if EA = 1, interrupts are enabled & will be responded to if bits in IE are high.

↳ if EA = 0, interrupts are disabled, <sup>not</sup> responded.

Example at Page 430.



## Timer interrupts

↳ Timer Flag (TF) is raised when timer rolls over

### Problem in Polling:-

↳ We have to wait until TF is raised ~~so that~~  
so, microcontroller is tied down while waiting  
for raising of TF, and can't do anything else.

How Interrupts solve this problem?

↳ If timer interrupt in IE register enabled,  
whenever time rolls over (TF is raised) & MC is  
interrupted in whatever it does, and jumps to  
interrupt vector table to serve ISR.

↳ so that MC can do any thing until it is  
notified that TF is raised.

see examples from 432 to 436 at slides

## External Hw Interrupts

P3.2 & P3.3  $\Rightarrow$  INTO & INT 1

\* activation levels for external Hw interrupts

↳ level triggered

↳ Edge " "



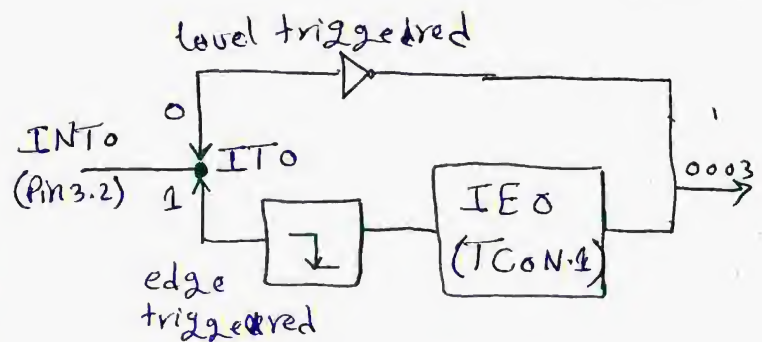
## Level triggered mode

↳  $INT0$  ( $INT1$ ) normally high.

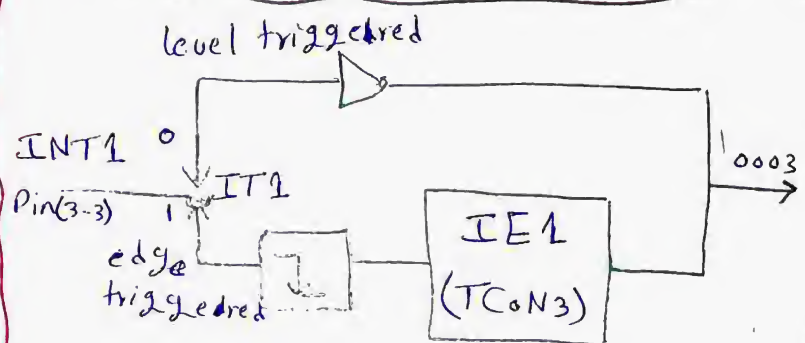
↳ If low-level signal is applied to them, it triggers interrupt.

↳ then  $\mu C$  stops whatever it's doing & jump to interrupt vector table to serve this interrupt.

↳ low-level signal at  $INT$  pin must be removed before executing last instruction of ISR ( $RETI$ ); otherwise another interrupt will be generated



Activation of  $INT0$

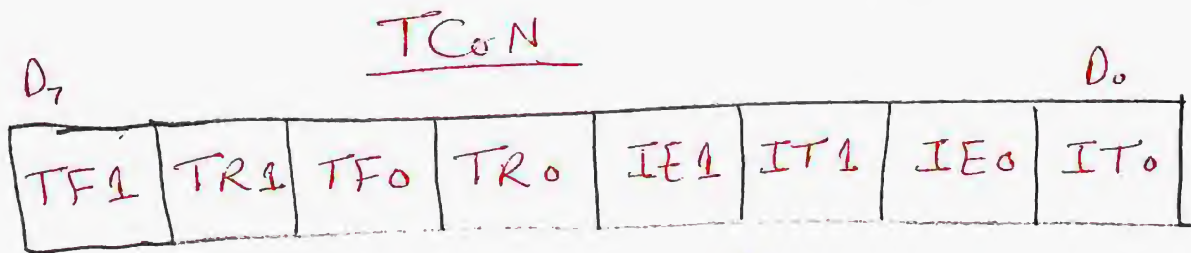


Activation of  $INT1$

See examples at Page 440

Note Pins P3.2 & P3.3 are used for normal I/O unless  $INT0$  &  $INT1$  bits in IE are enabled

\* To make  $INT0$ ,  $INT1$  edge-triggered interrupts  
↳ we must program bits of TCON register.  
↳  $ITO$ ,  $IT1$  are bits D0 & D2 of TCON register.  
↳ They are referred to as  $TC0N.0$  &  $TC0N.2$   
Since TCON is bit addressable.



\* TF1 (TCoN.7) → timer 1 overflow.

- ↳ set by HW when timer/counter 1 overflows.
- ↳ cleared by HW.

TR1 (TCoN.6) timer 1 run control bit.

- ↳ set/cleared by SW turn timer/counter 1 on/off.

TF0 (TCoN.5) → timer 0 overflow.

TR0 (TCoN.4) → ~~timer~~ timer 0 run control bit

IE1 (TCoN.3) → External interrupt 1 edge flag.

- ↳ set by CPU when external interrupt edge detected.
- ↳ cleared by CPU when interrupt is processed.

IT1 (TCoN.1) → interrupt 1 type control bit

- ↳ set/cleared by SW to specify falling edge/low-level triggered external interrupt.

see example at Page 446



### Notes

At edge-triggered interrupts:-

↳ external source must be held high for at least one machine cycle & then held low for at least " " " " .

↳ Falling edge of  $INT0$  &  $INT1$  are latched by 8051 & are held by  $TCON.1$  and  $TCON.3$

EX what is the difference between  $RET$  and  $RETI$  instructions? How we can't use " instead of  $RETI$ ?

↳ Both perform same action of popping of two bytes of stack into PC, and marking 8051 return to left off

But,  $RETI$  perform additional task of clearing the interrupt-in-service flag

If you use  $RET$  instead of  $RETI$  in last instruction of  $ISR$  → you blocks any new interrupt on that pin after first interrupt, cause  $RET$  don't clear interrupt in service flag.



## Serial Communication Interrupt

### TI (transfer interrupt)

- ↳ raised when last bit of framed data (stop bit) is transferred
- ↳ indicating that SBUF register read to transfer next byte.

### RI (received interrupt)

- ↳ raised when entire frame of data including stop bit is received.

In 8051 there is one interrupt for serial communication.

- ↳ used to send & receive data.

if IE.4 enabled, when RI or TI is ~~used~~ raised, the 8051 get interrupted and jump to memory location 0023H to execute ISR.



\* Serial interrupt is mainly used for receiving data & is never used for sending.

See examples From Page 453 to 458

## Interrupt Priority

↳ when 8051 is powered up, the priorities are assigned according to:

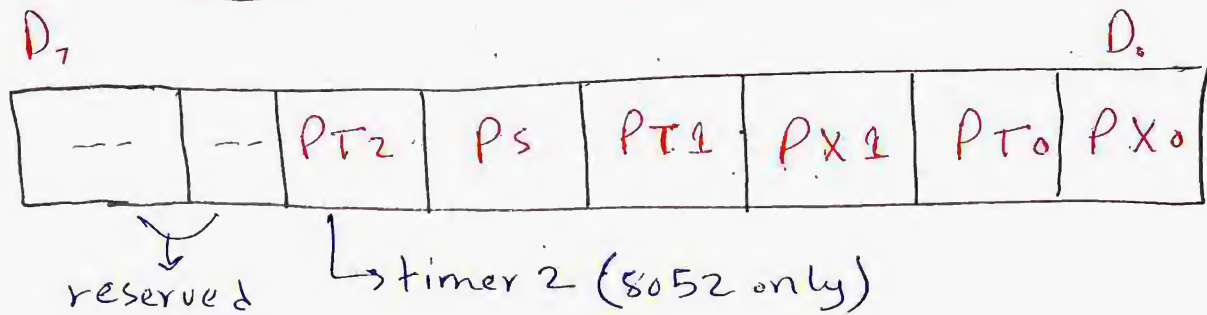
Highest to lowest:			Priority	Interrupt Priority upon reset
External interrupt 0			INT0	
timer	"	0	TFO	
External	"	1	INT1	
timer	"	1	<del>TF</del> TF1	
Serial Communication			RI + PTI	

in Page 461 example ask what if more than one interrupt activated at same time → answer will be this priority.

IP (interrupt priority) used to give higher priority to interrupts, we make bit in IP register → high.

→ if you have two or more interrupt bits in IP register high ⇒ you will go to the priority table.

## Interrupt Priority register



See examples at page 464 (465 (important))

Note in 8051: low priority interrupt can be interrupted <sup>by</sup> ~~but~~ higher " " not by another low - " " .

## Programming in C

→ Compiler C

\* ~~They~~ can assign unique number of 8051 interrupt,

\* It can assign register bank to an ISR

interrupt-name	number
INT0	0
TF0	1
INT1	2
TF1	3
RI+RI	4
TF2	5

Examples at page 469 (470)